Executable Modelling of Dynamic Software Product Lines in the ABS Language

Radu Muschevici
iMinds-Distrinet, KU Leuven

Dave Clarke
iMinds-Distrinet, KU Leuven
and Dept. Inf. Technology,
Uppsala University

José Proença
iMinds-Distrinet, KU Leuven
and HASLab/INESC TEC,
Universidade do Minho

ABSTRACT

Dynamic software product lines (DSPLs) combine the advantages of traditional SPLs, such as an explicit variability model connected to an integrated repository of reusable code artefacts, with the ability to exploit a system's variability at runtime. When a system needs to adapt, for example to changes in operational environment or functional requirements, DSPL systems are capable of adapting their behaviour dynamically, thus avoiding the need to halt, recompile and redeploy. The field of DSPL engineering is still in formation and general-purpose DSPL development languages and tools are rare. In this paper we introduce a language and execution environment for developing and running dynamic SPLs. Our work builds on ABS, a language and integrated development environment with dedicated support for implementing static software product lines. Our ABS extension advances the scope of ABS to dynamic SPL engineering. Systems developed using ABS are compiled to Java, and are thus executable on a wide range of platforms.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Dynamic Software Product Lines, Dynamic reconfiguration, Runtime adaptation, Delta modelling

1. INTRODUCTION

Various application domains require systems that run continuously and without interruption. This is especially true of highly-available applications such as mission critical software. At the same time, user requirements as well as conditions in the environment in which the system runs can vary over time. Such scenarios motivate the adoption of systems that can react to external variations by adapting themselves dynamically, in order to exhibit optimal performance and functionality under all conditions, while also eliminating upgrade-related downtime. One approach proposed to address this problem is dynamic software product lines (DSPL), which support the generation of system variants at runtime [3]. In other words, a deployed DSPL system that behaves as a certain product can be *reconfigured* (i.e., *adapted*) to behave as a different, valid product without the need to halt the system, recompile and redeploy.

While programming language support for static software product lines has been investigated extensively [23, 2, 27], comparable support for dynamic product lines is less well explored [3]. Dynamic reconfiguration is typically considered from the high-level perspective of feature or component modelling [16, 4, 22]. Among the methodologies for developing dynamic SPLs, *delta-oriented programming* (DOP) [27] is a relatively recent proposal. DOP handles changes to the set of selected features by triggering the application of deltas; this results in a program transformation, as deltas can add, modify and also remove code. The lower level support required for such dynamic transformations is available [18, 21, 30] and some of this has crept into features relevant for SPL engineering [10]. However, we do not know of any tool and language support for DOP-based DSPL.

This paper addresses the problem in the context of the Abstract Behavioural Specification language (ABS) [31], a recent specification and programming language developed in part by the current authors. ABS was chosen because it already offers comprehensive support for (static) SPL development based on DOP. The contribution of this work is threefold. First, we introduce support for modelling dynamic variability in the ABS language. Second, we introduce MetaABS, a reflective layer for the ABS language that enables introspection and manipulation of the running program from within itself, effectively enabling auto-adaptive models. Third, we develop a runtime engine that supports changing a running system incrementally, by applying deltas and state transfer functions, effectively modifying the system's structure (interfaces, classes, etc.) and objects, and also supporting changes to the variability model itself.

Static and dynamic product configuration in ABS differ in two key aspects: First, static product configuration always starts with the base product and applies a sequence of modifications until obtaining any of the products specified by the product line. Dynamic product reconfiguration starts with any product already configured using the above process, and applies a set of modifications to obtain a new product (out

FOSD '13, October 26, 2013, Indianapolis, IN, USA

ACM 978-1-4503-2168-6/13/10 ...$15.00
http://dx.doi.org/10.1145/2528265.2528266.

of a specified set of valid products). The second aspect is the necessity to adapt the program's runtime *state* in addition to adapting its structure.

The paper is organised as follows. Section 2 provides the necessary background on the ABS language. The rest of the paper presents our contributions. Section 3 details the ABS language elements for modelling dynamic SPLs and describes auto-reconfiguration using MetaABS. Section 4 describes the back-end implementation that enables the concurrent reconfiguration of systems at runtime. Section 5 explains how ABS supports openly adaptive models. Section 6 discusses related work and Section 7 draws our conclusions and highlights future work.

2. BACKGROUND: THE ABS LANGUAGE

The ABS language allows the precise modelling and analysis of concurrent systems, focusing on their functionality while abstracting from concerns such as concrete resources, deployment scenarios and scheduling policies [20]. At the same time ABS provides modelling concepts for specifying variability incrementally from the level of feature models down to object behaviour [7, 1]. Exploiting this variability is, however, until now limited to compile time. This section briefly introduces ABS, focusing on its concurrency model and its variability mechanisms.

2.1 Concurrency Model

The concurrency model of ABS is based on active objects, asynchronous method calls, and futures. Asynchronous method calls trigger concurrent activities, as both the calling and the called methods run in parallel. Futures enable the calling process to later retrieve the result of an asynchronous computation. In ABS, concurrent objects can be collected in concurrent object groups (COGs) [20]. COGs are active runtime entities possessing their own thread; objects inside the same COG share a common thread, scheduler and message queue—a COG is thus a locus of concurrency control of a collection of objects. Methods execute as tasks inside a COG and use cooperative multitasking, meaning that they release control of the thread only at designated points, using await and suspend statements. An `await(guard)` statement causes the process to suspend until the guard is true. Suspension of a process give another process in the same COG the opportunity to run. Naturally, control of the thread is relinquished whenever a method finishes.

2.2 Variability

Software variability is modelled in ABS following common SPL engineering practice [6]. This includes a *feature model* that defines the system variants abstractly using features and feature attributes, and a set of code components called *deltas* that implement units of variable behaviour. The two are connected via a product line configuration description that specifies for which feature combinations each delta is applicable (details omitted here). A *product* is defined in terms of a set of selected features and attribute values. For an in-depth introduction to the variability modelling capabilities of ABS we refer to Clarke et al. [7].

The delta-oriented programming paradigm [27] is a software development approach in which program variants are derived from a core program by applying a set of structural program transformations called deltas. The core defines the set of classes and interfaces that form a base product. Deltas express the addition, removal, or replacement of program elements such as classes, interfaces, methods, and fields.

```
module Chat;
interface Client { ... }
interface Text extends Client {
  Unit message(Client client, String msg);
}
class ClientImpl implements Client, Text {
  Unit message(Client client, String msg) { ... }
}
```

Figure 1: A core module of the chat SPL

As a simple example we introduce an SPL of chat applications, which will be extended in the following section to a dynamic SPL. The core of the chat system (Figure 1) consists of the classes needed to construct a chat product that only supports text-based communication. Variants of this system add support for voice and video communication, the necessary code being supplied by deltas. Figure 2 shows a delta `DVoice` that adds voice functionality by introducing new classes and interfaces, and modifying the core implementation of the class `ClientImpl`.

```
delta DVoice; // modify core to add voice functionality
uses Chat;
adds interface Voice extends Client {
   Call call(Client client);
}
modifies class ClientImpl adds Voice {
   adds List<Call> ongoingCalls;
   adds Call call(Client c) { ... }
}
adds interface Call { ... }
adds class CallImpl implements Call { ... }
```

Figure 2: Definition of delta modules

3. EXTENDING ABS FOR DYNAMIC RECONFIGURATION

This section describes the extensions to the ABS language to support dynamic product lines. How these are compiled and the runtime support for them are described in Section 4. A DSPL in ABS is a set of software products that are available at runtime, together with a reconfiguration decision model that describes the variability of the system at runtime as a set of reconfiguration steps. A *reconfiguration* takes place between two products and adapts the current product's structure and state (Figure 3). The possible reconfigurations between products are specified in a *reconfiguration decision model* (Section 3.1). The reconfiguration is performed by dynamically applying *deltas* (Section 3.2), and the state of objects is transformed according to state transformation functions declared in a *state update* (Section 3.3). As a result of reconfiguration, a different product becomes active and the system behaves according to the specification of the new product. That new product can be adapted into yet another product and so forth. Throughout the paper we refer to the product that is about to be adapted as the *current* product and the product obtained after adaptation as the *target* product. Each DSPL specification needs an *initial*

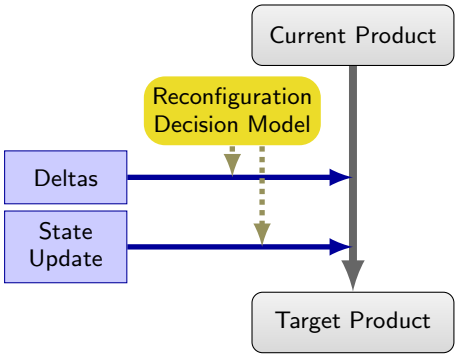

Figure 3: Elements involved in a reconfiguration

product, which is the product that has been validated and configured statically from the core and is active when the system is deployed. Reconfiguration of a DSPL can be initiated within the product line using auto-reconfiguration code written in the ABS meta-language, MetaABS (Section 3.4).

3.1 Reconfiguration Decision Model

The reconfiguration decision model defines the possible reconfigurations between products and how they are carried out. A reconfiguration is performed between two variants of the DSPL. The products are declared, as in the static SPL setting, by associating the product name with a set of features from the feature model. Additionally, each product declaration lists the possible target products of the SPL that the given product can be transformed into, together with a sequence of deltas and a state update.

The reconfiguration decision model for the chat SPL example (Figure 4) defines three products, by stating—for each product—the product's name and features, and the set of other product that it can be reconfigured into. The "low-end" chat product (line 1) implements only the `Text` feature. This product can be reconfigured at runtime into a "regular" chat product (line 4) that additionally supports the `Voice` feature by applying the delta `DVoice` and the state transfer function `L2R`. The third product is a "high-end" chat system that also supports video and file transfer (line 8). Both the static configuration options for the chat SPL and its reconfiguration decision model can be readily visualised (Figure 5).

```
1  product LowEnd (Text) {
2      Regular delta DVoice stateupdate L2R;
3  }
4  product Regular (Text, Voice) {
5      HighEnd delta DVideo,DFiles stateupdate R2H;
6      LowEnd  delta DNoVoice stateupdate R2L;
7  }
8  product HighEnd (Text, Voice, Video, Files) {
9      Regular delta DNoFiles,DNoVideo stateupdate H2R;
10 }
```

Figure 4: Reconfiguration decision model of the chat DSPL

3.2 Deltas

Transforming a software product with a certain set of features into a product with a different set of features generally requires changing both its structure and behaviour. For an ABS model this entails adding, removing or modifying model elements such as classes, interfaces, functions and data types. As in the static setting, this is done by applying a sequence of *deltas* to a core program, except that now deltas are applied while the system is running.

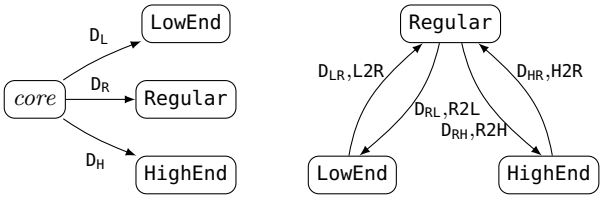

Figure 5: Left: static chat product configuration, right: dynamic chat product reconfiguration

The sequence of deltas necessary for each reconfiguration needs to be declared by the developer. For example, the reconfiguration decision model for the Chat product line (Figure 4) shows that the `LowEnd` product needs to apply the `DVoice` delta when adapting to the `Regular` product (line 2).

In general, this manual approach requires some overhead from the developer, who needs to declare additional deltas (in addition to the deltas used for static reconfiguration) and specify their order of application. For the chat SPL example, the three deltas `DNoVoice`, `DNoVideo` and `DNoFiles` (Figure 6) specify the removal of the `Voice`, `Video` and `Files` features. Deltas defined for static product configuration can be also applied dynamically, as is the case when reconfiguring the `Regular` product into the `HighEnd` product by using deltas `DVideo` and `DFiles`.

```
delta DNoVoice;
uses Chat;
removes interface Voice;
modifies interface Client removes Voice;
modifies class ClientImpl {
   removes CallHistory callHistory;
   removes Call call(Client client);
}
removes interface AudioStream;
removes interface Call;
removes class CallImpl;

delta DNoVideo; ...
delta DNoFiles; ...
```

Figure 6: Deltas used in the reconfiguration decision model (Figure 4)

3.3 State Updates

When reconfiguring a running system, its execution state, namely the collection of values assigned to variables and fields, needs to be preserved or adapted to the systems new structure. The challenges are how to adapt state elements to match the updated system, and to when to adapt state elements without disrupting the runtime execution.

While fully automated state update has been the focus of recent research [12, 24], the transfer of state information typically requires some manual guidance from the developer

in cases when state variables need to be mapped to new variables by a function more complex than simple identity. For example, if a field is removed, its value might need to be preserved by transferring it to a new field of possibly different type. Similarly, if a new field is added, it needs to be given sensible value, which may not be the default value.

We adopt a hybrid approach, automating the simple cases (e.g., fields that are present in the old and new code are carried over unaltered), combined with the ability for the user to manually define the transfer function for more complex scenarios. An ABS *state update* specifies *how* to transfer the values of fields to the object's post-reconfiguration state while also mandating *when* it is safe to do so.

A state update is a collection of *object updates*, each describing how to update objects from a given class. More precisely, an object update consists of: (1) the name of the class whose instances are targeted by the update, (2) an *update guard* mandating when the state update can be applied, (3) a set of declarations of local variables and functions used within the body of the object update, (4) a **classupdate** statement that triggers the update of the object's class (thus updating its interface and fields), and (5) a set of assignments used to initialise the object's fields, possibly based on values of its state before updating the class. When an object update is applied to update an object, the following steps are performed: The update guard is expressed by an ABS **await** statement, allowing the developer to specify when it is safe to apply the state update. The code before the **classupdate** is run in the context of the object's original class and is used to salvage values of fields that are removed by the update. The code after the **classupdate** is used to initialise added fields, possibly with values computed in the pre-update step. Note that the values of fields that are not affected by the state update (i.e. present in the old and new state) are carried over automatically. The names in scope before and after the **classupdate** are thus different. The scope changes according to how the object's class structure changes. For instance, if fields are removed, their names are not available after the class update. Similarly, if new fields are introduced by the new version of the class, they are only available after the class update. Variables defined locally within the object update are visible in both scopes. Applying an object update triggers the creation of a task that is scheduled to be executed on the COG where the object resides. This task can execute only when the update guard becomes true. The details of the implementation will be described in Section 4.2.

```
stateupdate R2L;
objectupdate ClientImpl {
  // pre–update section
  await(length(ongoingCalls == 0));

  def ChatHistory mergeHistories(CallHistory calls,
    ChatHistory chats) = ...
  ChatHistory mergedHistory;
  mergedHistory = mergeHistories(callHistory,chatHistory);

  classupdate; // change in scope
  // post–update section
  chatHistory = mergedHistory;
}
```

Figure 7: A state update declaration

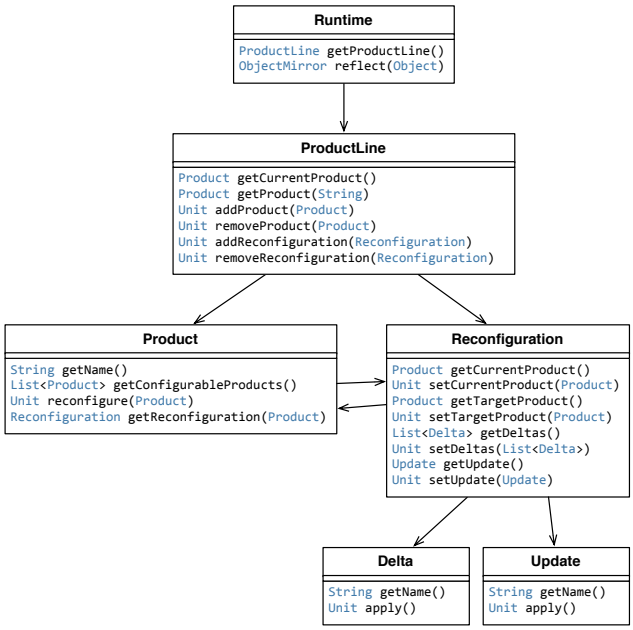

Figure 8: MetaABS API exposing the DSPL

An example state update (Figure 7) is used to adapt a "regular" chat software to the "low-end" variant. The update of chat clients (objects of class `ClientImpl`) is guarded by a condition that requires that no calls are ongoing with the client involved. We assume that the state of the regular client includes a history of calls and a list of chat sessions, stored in the fields `callHistory` and `chatHistory`, respectively. The `chatHistory` is common to both products, therefore its value is preserved by default. However, the history of calls will be lost upon removing the `callHistory` field, as directed by the delta `DNoVoice` (cf. Figure 6). To keep this information, both histories are merged into in a local variable `mergedHistory` in the pre-adaptation phase. In the post-adaptation phase this value is assigned to the `chatHistory` field.

3.4 MetaABS Support for Auto-Adaptation

To support product line adaptation autonomously at runtime, ABS introduces a dynamic meta-programming facility, called MetaABS, based on reflection. MetaABS exposes basic elements of the runtime environment to the programmer, enabling their inspection and modification. Among these elements are the running system's underlying DSPL structure, giving the running system the capability to reconfigure itself.

The part of the MetaABS API relevant for the dynamic reconfiguration of products (cf. Figure 8) centres around a `ProductLine` object, obtained by calling `getProductLine` on the `Runtime` object. The `ProductLine` interface gives access to ABS methods to obtain the current product and the products that can be obtained through reconfiguration of the current product, as defined by the reconfiguration decision model. Beyond simple introspection, it provides a `reconfigure` operation to actually reconfigure the current system to behave as a given target product. Furthermore, MetaABS makes products, reconfigurations, deltas and state updates accessible as objects of the language.

An example of how the developer can use MetaABS to

implement a global triggering of reconfiguration of the chat DSPL is given in Figure 9. The code monitors certain variables in its operating environment and adapts autonomously as these variables change. The reconfiguration logic is encapsulated in a `Reconfigurator` class. A reconfigurator instance runs as a separate process (i.e., concurrently to the chat functionality), monitors the network connection and transforms the running product depending on the available bandwidth. The highlighted calls invoke methods from the `ABS.Meta` library.

```
module Monitor; import * from ABS.Meta;
data Bandwidth = Low | Mid | High;
interface Connection { Bandwidth checkBandwidth(); }
class Reconfigurator(Connection conn) {
   Unit run() {
      ProductLine pl = getProductLine();
      while(True) {
         Product currentP = pl.getCurrentProduct();
         Product targetP;
         Bandwidth bw = conn.checkBandwidth();
         if (currentP.getName() == "RegularChat") {
            if (bw == Low) {
               targetP = pl.getProduct("LowEndChat");
               currentP.reconfigure(targetP));
            } else if (bw == High) {
               targetP = pl.getProduct("HighEndChat");
               currentP.reconfigure(targetP));
            }
         } else if (p == "HighEndChat") {
            if (bw == Low || bw == Mid) {
               targetP = pl.getProduct("RegularChat");
               currentP.reconfigure(targetP));
            }
         } else if (p == "LowEndChat") {
            if (bw == Mid || bw == High) {
               targetP = pl.getProduct("RegularChat");
               currentP.reconfigure(targetP));
            }
         }
      }
   }
}
```

Figure 9: Implementing runtime product reconfiguration for the chat product line using MetaABS

4. DYNAMIC RECONFIGURATION

Dynamic software product reconfiguration has two phases: changing the system structure and behaviour by applying deltas, and subsequently mapping the old execution state to the new system structure. The ABS user controls the application of both deltas and state updates via the reconfiguration decision model. We now describe the mechanisms that enable dynamic reconfiguration in ABS, that is, the mechanisms that delta and state update application rely upon.

4.1 Back-End Variability Support

We designed an adaptive runtime environment and an ABS compiler back-end that generates adaptive Java code. These tools are implemented and available as part of the ABS tool framework.[1] The key idea behind the dynamic Java back-end is to use dynamic structures in the target language to represent ABS elements. In practice, Java objects and the singleton design pattern are used to represent interfaces, classes, methods, objects and object fields. An example shall illustrate this setup. Adding a new class to the system is a common activity when configuring a new product. The new class is represented as an instance of the class `ABSDynamicClass`, which is provided by the backend. Fields and methods of the new class are also encoded as objects, and are associated with the class by calling the `addField` and `addMethod` on the class instance. Modifying an existing class amounts to adding and removing fields and methods by calling `addField`, `addMethod`, `removeField` and `removeMethod`.

[1] `http://tools.hats-project.eu/spl/dynamic.html`

4.2 Concurrent Reconfiguration

The runtime updating mechanism uses ABS's own concurrency model based on asynchronous method calls and cooperative multitasking. Like methods, updates are scheduled as tasks, in response to system reconfiguration requests (typically the MetaABS operation `Product.reconfigure`). In order to control when updates are applied to specific objects, a standard **`await`**`(guard)` statement is used. The guard defines the quiescence condition for each object. As long as the condition is false, the update task suspends; when true, the update is executed to completion. This gives the ABS developer the power to formulate what is considered a safe state for the objects of each class, and it ensures the update is performed when the object is in such a state.

In the following, our concurrent reconfiguration mechanism for ABS is presented in detail. Reconfiguration corresponds to globally modifying the system's structure by applying a sequence of deltas, and incrementally updating each object by applying a state update (cf. Section 3).

The Object Roster.

A state update contains *object updates* that define the reconfiguration of objects of a certain class. Prior to scheduling an object update, the system needs to obtain the set of objects in the system to which the object update applies. For this purpose, the runtime maintains the object roster, a set of objects for each class. In our implementation the roster has weak references, allowing the JVM garbage collector to collect objects that are no longer in use. The object roster is cleared periodically of references to objects that have been collected.

Sequencing Object Updates.

Object updates are applied in the order in which they were deployed, that is, no update can overtake one that was triggered by an earlier reconfiguration request. To ensure this, updates and objects bear version numbers. To apply an update to an object, their versions must be equal. Objects increment their version number after having been updated. Technically, if an update process is scheduled to run and the versions do not match, the process suspends. Consequently, the next-in-sequence update process gets a chance to run.

4.2.1 Global Reconfiguration Scheme

A product is reconfigured in two steps, as illustrated by the algorithm in Figure 10. First, a sequence of deltas `D1..Dn` is applied to a copy of the targeted classes, extending the system's class structure. These new classes will be linked to the running objects in the second step. Secondly, all

objects affected by structural changes (that is, all objects whose class was modified) are scheduled for update. The application of deltas and updating of objects are performed using MetaABS operations.

```
Require: deltas D1..Dn; state update U
  for all deltas D in D1..Dn do
    D.apply()
  end for
  for all classes C targeted in state update U do
    for all instances obj of class C do
      ObjectMirror objm = reflect(obj)
      ObjectUpdate u = U.getUpdate(C)
      objm.scheduleUpdate(u)
    end for
  end for
```

Figure 10: Global reconfiguration schematic

The first `for` loop in Figure 10 shows the application of the delta sequence associated with a reconfiguration. Deltas are applied atomically for each class. To achieve atomicity, all class modifications are applied to a copy of the targeted class; the copy eventually replaces the original class. When a delta adds a class, the new class is created. In case of class removal the targeted class is marked for removal (which prevents creating new instances), but it is only removed when no more objects of that class exist in the system. The timing of delta application is therefore not critical.

4.2.2 *Object Updating Scheme*

Object updates are scheduled individually for each object as tasks of their respective COG. They are executed observing their application guards. This corresponds to the second `for` loop in the reconfiguration scheme (Figure 10). The object updating algorithm is broken down into individual MetaABS instructions as shown in Figure 11. Finally, objects that are not targeted by the state update but whose classes have been updated by a delta need to update their class reference to the new class version. For these objects, the compiler generates object updates implicitly, with empty pre and post transformations, and guard set to `True`; they are included in state update `U`.

```
Require: new class C; state update U
  await(this.version == U.version && U.guard)
  initialise local definitions
  run pre-classupdate body of U
  Class c = this.getClass()
  Class newC = c.getNextVersion()
  this.setClass(newC)
  run post-classupdate body of U
  this.version += 1
```

Figure 11: Per-object reconfiguration schematic

Object updates are executed after a user-defined condition (`guard`) becomes `True` and the object matches the version of the update. The update consists in first saving elements of the old state that need to be preserved, then updating the object's class pointer to the new class version (created through delta application), and then initialising the new state, possibly based on values saved from the old state. Finally, the object's version is incremented.

The `R2L` state update (Figure 7), for example, is converted into the task presented in Figure 12. This example (arbitrarily) assumes that the update's version number is 2. The task is scheduled for every instance o of `ClientImp`, in the same way if it was a method of o being invoked asynchronously.

```
{ await(this.version == 2 && length(ongoingCalls == 0));

  // pre:
  mergedHistory = mergeHistories(callHistory,chatHistory);

  // class update
  Class c = this.getClass();
  Class newC = c.getNextVersion();
  this.setClass(newC);

  // post:
  chatHistory = mergedHistory;

  // version update
  this.version += 1; }
```

Figure 12: Task performing an object update

5. OPEN ADAPTIVITY

DSPL that can evolve at runtime by incorporating changes into their variability model are *openly adaptive* [3]. Such changes can be the addition, removal or modification of products or transitions between products. ABS supports open adaptivity by allowing changes to the reconfiguration decision model via the MetaABS language. The only restriction we impose is that the currently running product cannot be removed when the reconfiguration decision model is updated. Should this become necessary (for example if the evolved set of products is disjoint from the current set), then the adaptation has to be performed in two steps.

Adding variants to an existing DSPL requires injecting the corresponding code (i.e. products, reconfigurations, deltas and state updates) into the system at runtime. Our system provides a runtime interface that allows dynamic loading of code via Java's standard class loading mechanism.

The MetaABS API (introduced in Section 3.4) provides operations to add and remove products as well as reconfigurations, that is, transitions between products. It also supports modifying existing products by adding or removing reconfigurations, and re-setting a reconfiguration's state update and delta sequence. Figure 8 shows the MetaABS interface complete with meta-variability operations.

The model a programmer should have in mind when thinking about open adaptivity is that of a transformation between reconfiguration decision models. For instance, Figure 13 shows how the chat SPL could evolve by adding a `Premium` product that implements video conferencing. The implementation of this evolution is provided in Figure 14.

6. RELATED WORK

Our work relates to research in the fields of dynamic software product lines and dynamic software updating.

Dynamic Software Product Lines.

A large portion of DSPL research is concerned with the delineation of the field itself (principles, properties, chal-

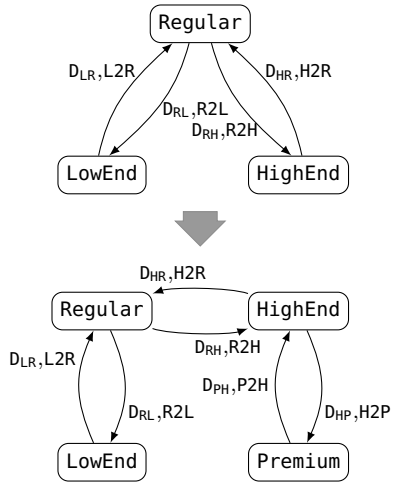

Figure 13: Evolving the chat DSPL

```
ProductLine pl = getProductLine();
Product high = pl.getProduct("HighEnd");
Product prem = new Product("Premium");
pl.addProduct(prem);
Reconfiguration h2p = new Reconfiguration();
h2p.setCurrentProduct(high);
h2p.setTargetProduct(prem);
h2p.setDeltas(...); h2p.setUpdate(...);
Reconfiguration p2h = new Reconfiguration();
p2h.setCurrentProduct(prem);
p2h.setTargetProduct(high);
p2h.setDeltas(...); p2h.setUpdate(...);
pl.addReconfiguration(h2p)
pl.addReconfiguration(p2h)
```

Figure 14: Adapting the reconfiguration decision model of the chat DSPL: MetaABS implementation

lenges) [14, 29, 19]. Our understanding of DSPL concepts is largely based on this research, but we focus on providing a language and tool implementation that support the development of DSPL. Context-oriented programming [8] and dynamic aspect-oriented programming [11] models have been used in connection with feature models as variability mechanisms for DSPL. Feature- and delta-oriented programming (FOP and DOP) have both been instrumented to support runtime (re)binding of features. Rosenmüller et al. [26] use FOP to statically compose sets of features called *dynamic binding units*, which can be switched on and off during runtime. Technically this is achieved by using the decorator design pattern to add behaviour to objects. A binding unit adds feature-specific behaviour by decorating the relevant classes. In contrast, we follow the DOP approach, which uses deltas to define (more general) program transformations. Our deltas exist and can be applied at runtime. We include a language feature that allows the developer to explicitly define how to transform the state of the program upon dynamic feature reconfiguration. DOP has been also applied recently to dynamic SPL [10, 9, 17] in the sequential setting of the DeltaJ language. In DeltaJ the reconfiguration space is formalised by an automaton and a `reconfigure` instruction specifies safe reconfiguration points in the program. In contrast, we describe the reconfiguration semantics in a concurrent setting based on active objects, where updates are incremental rather than global. Instead of quiescent program update locations, we use update-specific safety guards. Additionally we provide a concrete implementation.

Dynamic Software Updating (DSU).

DSU research focuses on the safety and timing of dynamic updates. Several DSU systems propose the automatic derivation of safety constraints for applying updates to single-threaded systems [18, 28]. When it comes to concurrent systems, such constraints may be both too broad to ensure timely application of updates, and generally insufficient to ensure safety [13]. This problem can be addressed by specifying safe update points in the code of each thread [25, 15]. Our approach is to specify safe update conditions attached to the update code itself; this is arguably more flexible because it allows the safety criteria to be tailored specifically to each update. Timing incremental updates is challenging from the perspective of preserving type safety when objects communicate with each other. Johnsen et al. [21] use type analysis to synchronise the updates of dependent objects. Wernli et al. [30] introduce first-class contexts that represent different versions of a system; these are kept mutually compatible with the help of bidirectional transformations. Our system currently lacks the ability to generally ensure type safety over the reconfiguration period. Another important challenge when building a DSU system is how to transfer the state when updating objects. Approaches range from simply preserving the values of unaltered fields and initialising new fields with default values [5, 28] to fully automated approaches [12, 24] based on analysing the program code or heap. Our solution automates the value transfer of unaltered fields but allows the user to specify state transformation functions.

7. CONCLUSION

This paper introduces an extension of the ABS language and tool suite that adds support for modelling, implementing and executing dynamic software product lines. This framework includes language constructs for specifying the runtime variability model, a meta-language to control reconfiguration from within the running model, and the implementation of both a runtime environment that supports runtime reconfiguration and a compiler that generates reconfigurable Java code. Concurrent systems developed in ABS are reconfigured incrementally, without the need for global quiescence.

Future work will address the issue of safe adaptation in the presence of concurrent objects. As distributed objects cannot be updated all at once without effectively halting the system, it is important to ensure that, in the course of an update, objects in an old state and those already in an updated state can interact seamlessly. Future work will also focus on automating tasks that currently require the user's intervention, such as the inference of deltas. We plan to extend the support for SPL model evolution by also representing the feature model at runtime. Finally, a case study and further practical evaluation will assess performance and scalability of dynamic SPL systems developed in ABS. We will also subject our tools to synthetic time/space efficiency measurements in order to evaluate and improve the efficiency of our approach.

8. ACKNOWLEDGMENTS

This work is supported by the FCT grant BPD/91908/2012. We thank the anonymous reviewers for their valuable feedback, which helped us improve the paper.

9. REFERENCES

[1] *The ABS Language Specification*, 2011. available at `http://tools.hats-project.eu/download/absrefmanual.pdf`.

[2] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In *GPCE*, pages 101–112. ACM Press, 2008.

[3] N. Bencomo, S. Hallsteinsen, and E. Santana de Almeida. A view of the dynamic software product line landscape. *IEEE Computer*, 45(10):36–41, Oct. 2012.

[4] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *DSPL*, pages 23–32. Lero, 2008.

[5] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE*, pages 271–281. IEEE Press, 2007.

[6] D. Clarke, N. Diakov, R. Hähnle, E. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.

[7] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In *FMCO*, volume 6957 of *LNCS*. Springer, 2011.

[8] P. Costanza and T. D'Hondt. Feature descriptions for context-oriented programming. In *DSPL*, pages 9–14. Lero, 2008.

[9] F. Damiani, L. Padovani, and I. Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *GPCE*, pages 1–10. ACM Press, 2012.

[10] F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *SPLC*, pages 34:1–34:8. ACM Press, 2011.

[11] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *Workshop on Composition and Variability*, 2010.

[12] C. Giuffrida and A. Tanenbaum. Safe and automated state transfer for secure and reliable live update. In *Workshop on Hot Topics in Software Upgrades*, pages 16–20, 2012.

[13] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[14] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *IEEE Computer*, 41(4):93–95, Apr. 2008.

[15] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster. A study of dynamic software update quiescence for multithreaded programs. In *Workshop on Hot Topics in Software Upgrades*, pages 6–10, 2012.

[16] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *DSPL*, pages 18–27. CMU, 2009.

[17] M. Helvensteijn. Dynamic delta modeling. In *DSPL*, pages 127–134. ACM Press, 2012.

[18] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, Nov. 2005.

[19] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *IEEE Computer*, 45(10):22–26, Oct. 2012.

[20] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.

[21] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *Formal Methods*, volume 5850 of *LNCS*, pages 596–611. Springer, 2009.

[22] J. Lee and K. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *SPLC*, pages 131–140. IEEE Press, 2006.

[23] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[24] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley. Automating object transformations for dynamic software updating. In *OOPSLA*, pages 265–280. ACM Press, 2012.

[25] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *PLDI*, pages 13–24. ACM Press, 2009.

[26] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel. Tailoring dynamic software product lines. In *GPCE*, pages 3–12. ACM Press, 2011.

[27] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, pages 77–91. Springer, 2010.

[28] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *PLDI*, pages 1–12. ACM Press, 2009.

[29] M. A. Talib, T. Nguyen, A. W. Colman, and J. Han. Requirements for evolvable dynamic software product lines. In *DSPL*, pages 43–46. Lancaster University, 2010.

[30] E. Wernli, M. Lungu, and O. Nierstrasz. Incremental dynamic updates with first-class contexts. In *Technology of Object-Oriented Languages and Systems*, volume 7304, pages 304–319, 2012.

[31] P. Y. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14:567–588, 2012.